Communications of the Association for Information Systems

Volume 2 Article 3

July 1999

Focus Issue on Legacy Information Systems and Business Process Change: Migrating Large-Scale Legacy Systems to Component-Based and Object Technology: The Evolution of a Pattern Language

A.J. O'Callaghan
DeMontfort University, aoc@dmu.ac.uk

Communications of the Association for Information Systems

Volume 2, Article 3
July 1999

FOCUS ISSUE ON LEGACY INFORMATION SYSTEMS AND BUSINESS PROCESS CHANGE:

MIGRATING LARGE SCALE LEGACY SYSTEMS TO COMPONENT- BASED AND OBJECT TECHNOLOGY: The Evolution of a Pattern Language

A. J. O'Callaghan
Faculty of Computing Sciences and Engineering
DeMontfort University, Leicester, United Kingdom
aoc@dum.ac.uk

LEGACY INFORMATION SYSTEMS AND BUSINESS PRESS CHANGE

NOTE: LETTERS TO THE EDITOR FOLLOWS PAGE 39

Click here for hperlink to the Letters Page

FOCUS ISSUE ON LEGACY INFORMATION SYSTEMS AND BUSINESS PROCESS CHANGE:

MIGRATING LARGE-SCALE LEGACY SYSTEMS TO COMPONENT-BASED AND OBJECT TECHNOLOGY: The Evolution of a Pattern Language

A.J. O'Callaghan
Faculty of Computing Sciences and Engineering
De Montfort University, Leicester, United Kingdom
aoc@dmu.ac.uk

ABSTRACT

The process of developing large-scale business critical software systems must boost the productivity both of the users and the developers of software, while at the same time responding flexibly to changing business requirements in the face of sharpening competition. Historically, these two forces were viewed as mutually hostile. Component-based software development using object technology promises a way of mediating the apparent contradiction.

This paper presents a successful new approach that focuses primarily on the architecture of the software system to migrate an existing system to a new form. Best practice is captured by software patterns that address not only the design, but also the process and organizational issues. The approach was developed through four completed, successful live projects in different business and technical areas. It resulted in a still-evolving pattern language called ADAPTOR (Architecture-Driven and Pattern-based Techniques for Object Re-engineering).

This article outlines the approach that underlies ADAPTOR. It challenges popular notions of legacy systems by emphasizing business requirements. Architectural approaches to migration are then contrasted with traditional reverse engineering approaches, including the weakness of reverse engineering in the face of paradigm shifts. The evolution of the ADAPTOR pattern language is

outlined with a brief history of the projects from which the patterns were abstracted.

Keywords: Legacy system migration, object-oriented development, component-based development, reverse engineering, software patterns, pattern languages

I.INTRODUCTION

The process of developing large-scale business critical software systems faces twin challenges as general pressures in the global economy make themselves felt in computing. One pressure is the need to continually boost the productivity both of the users and the developers of software; the other is to enable software systems to respond flexibly to changing business requirements in the face of competitive pressure. These two forces historically were viewed as mutually contradictory (increased productivity generally resulted in the past from centralization and economies of scale). Component-based software development utilizing object technology promises a way of mediating the apparent contradiction and delivering an attack on both fronts simultaneously [Graham 1995]. However, the movement of any large-scale business-critical system to components is fraught with difficulty. Each such system has a development history, intimately tied not only to its function but through that to a business and organizational context that is always unique. Even more difficult, therefore, is the problem of generalizing from successful practice in such a way as to reuse hard-won expertise and develop guidelines for the successful migration of numbers of such systems in an enterprise.

This paper presents a new approach, which has already demonstrated success. It focuses primarily on the architecture of the software system in order to migrate an existing system to a new form. Best practice is captured in the form of software patterns [Coplien 1996] that address not only the design, but also the process and organizational issues that inevitably surround such a project. The

approach was developed through five successful live projects in different business and technical areas. It resulted in a still-evolving pattern language called ADAPTOR (Architecture-Driven and Pattern-based Techniques for Object Re-engineering) [O'Callaghan 1998b]. Although developed in the telecommunications sector, ADAPTOR is currently being tested in both the machine-tools industry and the defence sector, suggesting wide applicability.

II. WHAT IS A LEGACY SYSTEM?

The *Oxford English Dictionary* defines a legacy as "a tangible or intangible thing handed down by a predecessor; a long-lasting effect of an event or process". Legacy information systems are typically the targets of reverse engineering projects. Legacy systems have been defined as stand-alone applications built during a prior era's technology [Ulrich 1994], but they are perhaps more widely understood as software systems whose plans and documentation are either poor or non-existent [Connall & Burns 1993]. More than fifteen years ago the size of the 'legacy problem' in the US was already estimated at US$400 million resulting from the labour of more than half a million IT professionals over the previous thirty years [Appleton 1983]. More recently it was suggested that organizations spend US$70 billion each year to maintain an estimated 10 billion lines and more of code [Lerner 1994].

The scale of this problem is simply not explained by the definition of legacy systems given by either Ulrich or Connall and Burns. To take the first, if new technology is available why have not all the 'old' systems simply been made redundant? As for the second, if the essence of the problem is poor or missing documentation, then why is the problem apparently worsening despite the infusion of ISO 9001 and other quality assurance mechanisms into the field of software development? It seems that the term ‘legacy system’ has become something of a catch-all for any installed system that has any kind of problem. Since there is no known general panacea that will 'cure' all these systems, it follows that only a subset of this wide spectrum of systems comprises genuine candidates for migration projects. The option of completely replacing an existing system is

always a possibility, but what is needed is a definition that aids in identifying likely candidate systems.

MAIN SOURCES OF THE 'LEGACY PROBLEM'

The area that can be usefully addressed is best scoped by looking at the business consequences of the recent history of IT deployment. The spread of the PC from the mid-1980s encouraged a culture in which 'point solutions' were developed. Departments, or even single business users within large companies, developed individual procurement and/or software development policies to meet their perceived needs, often on an application-by-application basis. Similar applications running on different operating systems on different boxes became common. Worse still, key business abstractions such as 'Customer' could be running on different applications on the same machine at the same time, and since these applications could not talk to each other, information integrity could not be maintained. Subsequently, such point solutions became subject to localized optimizations, and uncontrolled maintenance, exacerbating the position even further.

Meanwhile successful systems simply aged, some less gracefully than others. Jones estimates that the average rate of change of software systems is between 5% and 7% every year, year on year [Jones 1994]. The compound impact over a period of years is such as to degrade the original structure of the system in an increasingly uncontrolled way.

Most crucially of all, accelerated competition in the global marketplace rendered the corporate environment more volatile than ever before. Mergers, takeovers, shutdowns and corporate restructuring can turn well-planned, well-engineered up-to-date systems into obsolescences virtually overnight.

The combination of these factors lends an aspect of inertia to software or IT systems when compared to the required agility of the enterprise that owns it. These kinds of legacy systems are typically both large and complex. Bucken for example, describes an organization that owns more than 1000 software programs, with an average age of 17 years, containing an accumulated total of 140 million lines of COBOL source code [Bucken 1992].

A NEW, WORKING DEFINITION OF 'LEGACY SYSTEM'

Drawing on the characteristics of legacy systems described above a more useful definition of the term 'legacy system' is as follows:

"A legacy system is a large system delivering significant business value today from a substantial pre-investment in hardware and software that may be many years old. Characteristically, it will have a long maintenance tail. It is, therefore, by definition a successful system and is likely to be one that is, in its own terms, well-engineered. It is a business-critical system which has an architecture which makes it insufficiently flexible to meet the challenges of anticipated future change requirements" [O'Callaghan 1996].

This working definition was adopted by the Object Engineering and Migration (OE&M) group at De Montfort University more than two years ago to develop criteria to judge which so-called legacy systems were suitable for conversion to component-based or object-based architectures, and which required different actions altogether. The definition all but rules out the type of system described by Connall and Burns above. Crucially, it establishes the business case as the key criteria for determining whether or not to migrate a legacy system and identifies the software architecture as the main focus of attention for the migration process.

The argument for this last point is established in Section III. What we can say here, however, is that in examining the problem space we find that it is only in small part occupied by technology. It is a mistake to see the 'legacy problem' as purely a technical one. The essence of the problem is as much in the nature of business requirements as it is for a 'greenfield' development. If a company in the financial services sector, for example, wishes to sell new products but increasingly finds its current IT investment inefficient in responding to these new requirements then the issue is decidedly not "How do we preserve as much of the existing system as possible?" but rather, "How do we best support the sale of new financial products, and what is the optimum configuration of IT for this purpose?" The solution may indeed involve new software development to replace the old system, or it may mean simple, incremental enhancement of the old system, or a

mixture of both. The point is that the legacy problem is primarily a business problem, and only incidentally a technical one, and that any solution must be driven from the problem space. A business case must be made for each and every proposed legacy system migration before it can proceed. In this respect, legacy system migration can be considered as a form of forward engineering that is actually much closer to 'greenfield' development than it is normally considered to be, differing from it only in that it consciously 'reuses' some amount or other of legacy software.

III. ARCHITECTURAL APPROACHES TO MIGRATION VERSUS TRADITIONAL REVERSE ENGINEERING

The migration of legacy systems is a process of re-engineering. The accepted definition of re-engineering is that put forward by Chikofsky and Cross [1990, p. 14]: "the examination and alteration of the target system to reconstitute it in a new form".

There is a particular quality to the re-engineering effort that must be understood when it involves moving a computer system from, say, a structured representation to an object-based one, however. A strong research tradition utilizes *reverse* engineering techniques, typically based on formal methods, to achieve restructuring of a legacy system - perhaps to move it from one language representation to another. In its own terms this approach, in different variations, can be shown to have achieved significant successes [e.g. Lano & Houghton 1993]. However, when a shift is being contemplated from, say, representation in a structured language to representation in an object-oriented implementation, it is not just the language that is changing but the development paradigm itself. An examination of the roots of this notion leads us to the conclusion that the same is true for component-based development in general since most current notions of components assume object-like encapsulation [Sprott & Wilkes 1998].

OBJECT MIGRATION IS A PARADIGM SHIFT

Cook [Cook 1994, Cook & Daniels 1994] showed that object-oriented development is a completely different, and essentially alien, paradigm [Kuhn 1970] to the traditional methods that preceded it. Object systems are structured around modules that bundle data and process together. Objects encapsulate data by the operations that query and manipulate them [Meyer 1988, Booch 1991]. Such a structure separates the software solution at a given level of abstraction from the architecture of the underlying machine, which is, contrariwise, based on the strict separation of data and process. By contrast, structured methods retain that separation at even the highest levels, implicitly imposing the underlying architecture of the machine on the overall software solution [O'Callaghan 1994].

In effect, the use of object-oriented methods imposes a significantly different separation of concerns upon systems from those of the traditional paradigm. In process-rich 'structured' environments,

- the analysis and design methods used,
- the notation, and
- the available syntactical constructs of implementation languages

support an attack on complexity through algorithmic decomposition, typically through top-down, stepwise refinement [Wirth 1971]. The structured approach produces a levelled separation of concerns from the main function through its sub-functions, which is readily seen for example, in levelled sets of data-flow diagrams or in the canonical, hierarchical form of a structure chart [Constantine and Yourdon 1976]. In structured environments that are data-centred, the major separation of concerns is between the 'stable' data structures and the 'more volatile' processes that support them. Then the data structure itself is organized around representations of entities [Chen 1989], each uniquely described in terms of its attributes, and the relations between them. The essential separation of concerns is reflected in the entity relationship diagrams that are produced. It is the separations of concerns in a system, and the way they are represented in software, which is the basis of the software architecture of any system. Selic

warns that they should be regarded as the software equivalents of load-bearing frames in building architectures [Selic *et al.* 1995].

In contrast to the architectures of structured systems described in the last paragraph, an object system can be thought of as a network of individual 'virtual machines'. Each of these machines is responsible for manipulating its own data set, and the overall functionality of the system is delivered by them sending messages to each other, to invoke their special behavioural responsibilities. Object systems are ‘architecture-free’ in that crucial sense [Cook 1994]. They are relatively unconstrained by the machine's requirement to separate data and process. It means object-oriented developers are free to supply a software architecture that is more strongly shaped by the shape of the business problems they are solving than has ever been possible with structured methods.

Graham, therefore, speaks of object-orientation as a general method for knowledge representation [Graham 1998]. That is, objects in a run-time system can map onto abstractions of human knowledge about the real-world problems the system is trying to solve. Graham concurs with a long line of object theorists and practitioners [e.g., Meyer 1988, Cook 1994, Martin & Odell 1998]. More importantly, for legacy system migrations in the context of BPR, a software architecture that maps closely onto the key abstractions in the problem space offers two other key advantages:

- the possibility of being able to change at a rate close to the rate of change of the key abstractions themselves
- the likelihood of maintaining traceability from solution to requirements through such business-driven changes.

THE IMPORTANCE OF MODELLING THE PROBLEM SPACE

The relative failure of traditional reverse engineering techniques when applied to the restructuring of systems to an object-based or object-oriented form results from their tendency to ignore the changing problem space which, typically, is driving the need for change in the first place. They are concerned with changing the representational form of the system, which, as Brooks [1986, 1995] eloquently reminds us is an accidental (i.e. incidental) task far removed from the essential

tasks of dealing with conceptual complexities. While there is no doubt that changes in representational form from some structured programming language to some object-oriented one are possible using these techniques, there is serious doubt as to whether many, if any, business benefits result. For example, a recent report from Hong Kong of a migration using reverse engineering and design recovery techniques achieved a degree of technical success, but at a huge cost, probably not worth the effort [Liao *et al*. 1998, O'Callaghan 1998a].

The *raison d'être* for contemplating a move to an object-based representation for an existing system is the belief that business benefits in terms of increased flexibility to business change, and increased productivity (through software reuse) will result. But these benefits rely, as we have seen, on the fact that object systems 'break' from the underlying Von Neuman architecture of the machine and enable the possibility of building software solutions in the image of the problem space itself. This understanding demands an approach to the migration of legacy systems which focuses on the software architecture, and which follows as closely as possible the well-understood forward engineering techniques of object-oriented development, especially those of object modelling. Such an approach draws upon research gains in this area dating back to 1991 [Jacobson 1991] and developed further by the OE&M group at De Montfort University. A beneficial side-effect of such an approach, incidentally, is a reduced cost of legacy migrations since specialist 'reverse engineering' skills and tool sets are not relied upon, but rather many of the same skills and tools employed in 'greenfield' object development can be reused.

The arguments in favour of this architectural approach against a traditional reverse engineering approach to the migration of systems to objects are presented fully in O'Callaghan, 1997. The differences are summarized in Table 1.

Table 1 The Differences Between 'Architectural' and Traditional Reverse Engineering Approaches to Legacy System Migration

Architectural approaches	**Traditional approaches**
Seek to impose a new separation of concerns	Seek a translation from one representation to another
Heuristic, and informal methods	Formal methods
Regards the legacy system as 'living history'	Regards the legacy system as archaeology
Problem-centred	Solution-centred
Modelling is emphasized	Rule-based transformation is emphasized
Focus is holistic, with an emphasis typically on the human maintainers of the system (or their organization) for input	Focus is typically on source code for input
Typically multi-paradigm or cross-paradigm	Typically single paradigm

Armed with the theoretical understanding elaborated above, the OE&M group established a track record of success in legacy system migrations in the telecommunications sector since 1993. The same approach is currently being tested in two different sectors, the CAD/CAM industry and the defence industry. Those experiences, plus other successes that we are aware of have been abstracted in the form of a software patterns language, ADAPTOR. The evolution of this still-developing pattern language is described in Section IV.

IV. THE EVOLUTION OF THE ADAPTOR PATTERN LANGUAGE

A migration from a system built using, say, structured methods to a component or object-based architecture involves above all imposing a separation of concerns upon the system that is different from the one that it was originally designed to reflect. Since migration is always a costly and somewhat risky venture, it tends to be enterprises that rely on software systems for their day-to-day business operations which can present the necessary business case for such a shift. In most cases it is changing business requirements, and the need for such systems to be flexible and adaptable to them, which creates the need for a new

architecture. The worldwide telecommunications sector provides an exemplar of such needs.

WHY LEGACY MIGRATIONS TO 'COMPONENT ARCHITECTURES' ARE ATTRACTIVE TO THE TELECOMMUNICATIONS SECTOR

Telecommunications companies developed historically as either state-owned or private monopolies in each country (sometimes, as in the case of the UK, originally as part of a wider communications infrastructure such as that provided by the UK Post Office). However, the force of global competition led to deregulation and the rise of new start-ups, followed by mergers, takeovers, and international operations agreements as the new market takes shape. At the same time, new technology in cabling, cellular telephone traffic, satellite transmission, and digital broadcasting changed the very nature of the services some of the more traditional companies provide. Network services in the new millennium will include more of the numerous features such as call waiting and voice mail that are strictly related to their telephone operations, but will also involve other services such as video on demand. While start-ups can take advantage of the latest technology, the traditional service providers have a huge investment in software, hardware and peopleware in systems that are 10-20 years old already. Where it is either too expensive or too risky simply to replace such systems, they have to cater for new, unanticipated requirements if their owners are to remain business competitive.

Even the companies that have the luxury to be able to invest in greenfield systems must be prepared to regard their new systems as legacy systems almost immediately after installation. Component-based software architectures are therefore extremely attractive to telecommunications companies. It is in this sector that the OE&M group deployed the architectural approach to migrating legacy systems described above. Four successful projects have been completed since the summer of 1993. Except in the case of the first of these projects (discussed below), commercial confidentiality agreements prevent discussion in great detail of the systems involved or the companies that own them. However, Table 2 demonstrates that each project was in a different business area and had different

technical characteristics, despite the fact that all the projects lay in the same industrial sector, i.e. telecommunications.

The common characteristic of each of these systems (the case studies were 'live' subsets of each of them) was the need to impose a new separation of concerns upon them, i.e. a new software architecture, in response to changing business requirements. In each case the systems' owners made a business decision that they needed a component-based architecture in order to meet the challenge of ever new requirements, and this architecture implied the kind of encapsulation that object-based systems deliver. Note however, that the architecture did *not* necessarily imply an object-oriented *implementation*. Indeed, the first two pioneering projects delivered a restructured system in the same base technology in which the legacy system was originally implemented.

Table 2 Successful Telecommunications Sector Projects that Inform the ADAPTOR Pattern Language

Year	Business Area	Business Requirement	Technical platform of legacy	Target platform
1993-5	Customer service support	Flexible and configurable tax calculation	MVS/ COBOL	MVS/ COBOL
1996-7	Network services management	Flexible, extensible support for feature development	ANSI C, pre-ANSI C / Oracle (plus in-house scripting languages)	ANSI C /Oracle
1997-8	Resource management	Adaptable architecture	C / C++	C++
1997-8	Pricing and charging of network services	Components for reuse	C	C++

In all of these systems, irrespective of the target implementation technology, object modelling was used to capture a description of the existing system in its business context, describe the new architecture, and plan the technical migration. In each case the migration itself was carried out by the client's own developers, with the OE&M group playing a knowledge transfer role. The particular role of OE&M group members in the project team itself differed from

project to project. In two projects they had a hands-on part in software design, in another two as visiting consultants, primarily in the 'analysis' phase. All the projects were considered to be successful in terms of their immediate technical objectives, their medium to long-term business objectives, and in their strategic and tactical research objectives. With each new success, new insights were gained into what elements of the approach were common to all the projects, and what elements were genuinely specific to each of them.

From the beginning, an objective of the OE&M group was to distill generic guidelines for migrating legacy systems to object-based and/or component-based architectures so that the lessons could be 'downstreamed' for all the relevant developers in the host organization. The difficulties of supplying meaningful abstractions so that experiences of success could be communicated with clarity and efficiency were reported after the first project to two conferences in 1996 [Farmer *et al*. 1996, O'Callaghan 1996], but by this time design patterns had made their appearance in the object-oriented community. It was decided to test the possibility of framing the migration guidelines in terms of software patterns. The network services project begun in the spring of 1996 was designed specifically to establish the feasibility of using software patterns to communicate best practice experience in the migration of large-scale legacy systems to component-based architectures.

SOFTWARE PATTERNS

Although interest in patterns was at the time almost exclusively focused on greenfield object-oriented design, the OE&M group's review of the literature led to the conclusion that there was no *a priori* reason in theory to believe this necessarily had to be so. Moreover, much of the theoretical inspiration for patterns came from, as it still does, the architect of the built environment Alexander, who has authored 253 patterns for constructing gardens, rooms, buildings, towns and communities to express a new way of building [Alexander *et al.* 1977]. Alexander is driven by the contradiction he has long observed that traditional societies who had no architects and no engineering or scientific discipline of architecture as such were nevertheless far more successful at

building 'living structures' than is modern society [Alexander 1964]. One of Alexander's first contributions was to reject the modern split between architect (who theorizes) and builder (who constructs, following the architect's drawings) in favour of a combination of user (inhabitant)-centred design and an architect-builder model in which the architect also implements [Gabriel 1996]. The central mechanism for this ongoing partnership between the inhabitants of a living space or a working space and the architect-builder was the pattern language. One way of viewing a pattern language, then, is as a way of capturing and communicating best practice - the modern equivalent of the cultural vehicles of the traditional builders that Alexander so admires.

The pragmatic attractions of using patterns to capture and describe best practice in an *architectural* approach to the migration of legacy systems were ultimately many. They included the following:

- Patterns structure the solution according to the problem.
- They abstract and communicate successful solutions.
- They are usable in varied specific contexts.
- They are applicable to software systems in general, not just object systems (and should therefore be applicable to legacy systems).
- They resolve non-functional (i.e., architectural) forces.
- They can be regarded as microarchitectures in their own right.

From a more theoretical aspect the practice of the use of software patterns is strongly suggestive of some of the academic discussion current in the built environment. Patterns attest to architecture being both a product and a process. A building is, at its most elementary level, a construction of physical elements or materials into a more or less stable form, as a result of which a space is created that is distinct from the ambient space that surrounds it. Thus, every building is both a physical and spatial transformation of the situation that existed before the building was constructed. Bill Hillier argues that at each step complex logical and sociological transformations are involved as well as physical ones [Hillier 1996]. The space carved out by the building is physically separated from ambient space, but this itself implies a mutually interdependent relationship between the

logical notions of an 'inside' and an 'outside'. Moreover, the drawing of the boundary also establishes a *social* separateness of the protected space which is identified, typically through ownership, with an individual or group which claims special rights over it. Hillier points out that these complex sets of physical, logical and sociological relationships are implicit in every construction from a primitive shelter to the most complex skyscraper. He calls complex schema of such relationships 'configurations' and posits that the notion of architecture deals essentially with these configurational aspects.

For Hillier, configuration is non-discursive. By 'non-discursive' he means, we do not know how to talk about it. We can recognize or even use configurations long before we can put a name to them. Indeed the normative behaviour they generate seems to depend on their existence as abstract ideas at levels other than conscious thought. Analytical knowledge is deliberately articulated by science in order to put it at risk, so that it can be challenged by other theories and hypotheses while the very purpose of configurational ideas would be put at risk if articulated. Indeed we normally take configurations so much for granted that it is only when confronted with another, culturally distinct, set of configurational ideas that we often become aware of them. In the built environment the use of 'standard' configurations leads to vernacular building. In vernacular building the non-discursive aspects of building are handled autonomically and more or less unconsciously, but architecture begins when these concerns become the object of reflective, critical and creative thought, when "the designer is in effect a configurational thinker" [Hillier 1996, p. 46)

From this perspective, patterns in general, and software patterns in particular, may be considered to be a way of making the hidden, social knowledge of construction explicit.[1] Certainly, it is clear that patterns document what has typically gone undocumented previously. In many cases a pattern puts into literature what an expert developer considers second nature. On a number of occasions in our own experience, the presentation of a pattern would draw a

[1] Hillier chose the term 'configuration' over 'pattern' only because he thought the latter carried a connotation of regularity that he wanted to avoid.

response from an expert along the lines of "But that's always the way I do it!" or even "That's just commonsense", while in the same workshop a junior developer would respond along the lines of "I wish I had had that pattern when I was working on the *X* problem…", implying that the junior developer had not yet sufficient experience to acquire the configurational ideas that structured the creative thought of the expert.

Making these ideas explicit also puts them at risk. But in a paradigm shift, explicitness is necessary. The vernacular construction techniques of structured methods are not the same as the vernacular of object-orientation or of component-based development. Arguably, the cause of most failed migrations has been the failure to understand that a different mindset is required to build successfully with objects and/or components [O'Callaghan 1994]. This conclusion suggests that the systematic articulation and dissemination of the non-discursive aspects of object-oriented and component-based construction is a minimum requirement for success. If, indeed, patterns do successfully capture these aspects, then they can help short-circuit the otherwise lengthy learning curve that a migration and/or development team might need to go through to acquire this hidden social knowledge which can otherwise only be gained through hands-on experience and learning-by-doing. In short there were good pragmatic and theoretical reasons for having confidence in a patterns-based approach.

INITIAL EXPERIENCES IN THE USE OF 'MIGRATION PATTERNS'

In the first project that patterns were applied to (the network services system, 1996-7, listed in Table 2) the aim was to produce a small catalogue of loosely related design patterns, united by the fact that they were useful in migrating legacy systems. It was expected that this would include design patterns 'mined' from the legacy system, together with object-oriented patterns reflecting the new architecture, and, perhaps, some special 'transitional' patterns which would be specific to the migration phase. A technique we called 'pattern panning'[2]

[2] The term deliberately evokes analogies with gold prospecting. The more usual ways of abstracting patterns either from source code, or from interviews with domain experts is often called 'pattern mining' (Rising 1997). Pattern panning is

was utilized in which designs and design decisions were documented as they were made during the migration phase of the development project. These documents were then examined off-line to write candidate patterns. The candidate patterns were drafted according to a pattern template customized to meet the specific needs of the host organization's developers [Harries 1996] and then presented back for peer review to developers' workshops (sometimes special purpose, sometimes 'piggy-backed' on to the project's regular design workshops). They were then redrafted and presented to a final validation workshop, which included developers who were familiar with the system, but not directly involved in the migration project. Patterns that passed through this final validation were then included in a catalogue, which contained, besides the patterns themselves, a graphical pattern map showing the relationships between the individual patterns.

ORGANIZATION AND PROCESS PATTERNS

The project was successful in uncovering a number of patterns, more than was anticipated, and was highly successful in that regard. But there were also a number of unanticipated developments:

- the use of many more existing, public domain design patterns than was anticipated.
- the discovery of patterns similar to public domain patterns, but which addressed a different problem.
- the need to apply patterns to non-technical areas in the migration project (e.g. process, organization).
- the realization of a greater 'interconnectedness' between our patterns than was anticipated.

The need to address organizational and process issues arose when at one point a sound technical solution had to be dropped because it strayed into parts of the system under the ownership of a different development team in the host organization. Examination of the ownership issue showed that the software

---

more a form of participatory action research in which we extracted the pattern 'nuggets' from an actually flowing stream of developmental activity.

development team structure faithfully reflected the high-level architecture of the team itself. Changing the architecture challenged the ownership structure. This finding turned out to be an illustration of *Conway's Law*, a pattern included in Jim Coplien's pattern language for the software development process [Coplien 1997]. *Conway's Law* states that, over time, "organization follows architecture; or architecture follows organization". Once this was understood then the need to explain to our hosts that they had to deal with the organizational barriers to a successful technical migration led us to two ideas simultaneously:

- the idea of using patterns to describe the non-technological issues surrounding migrations.
- the possibility, not just of a patterns' catalogue, but of a pattern language for the migration of legacy systems.

PATTERN LANGUAGES

Gabriel, one of the members of the software patterns movement most influenced by Alexander, writes,

"A pattern language is a set of patterns used by a process to generate artifacts. These artifacts can be considered complexes of patterns. Each pattern is a kind of rule that states a problem to be solved and a solution to that problem. The means of designing a building, let's say, using a pattern language is to determine the most general problem to be solved and to select patterns that solve that problem. Each pattern defines subproblems that are similarly solved by other, smaller patterns. Thus we see that the solution to a large problem is a nested set of patterns" [Gabriel 1996, p. 46].

Schmidt, Fayed and Johnson similarly state that "When patterns are woven together they form a language that provides a process for the orderly resolution of software development problems" [Schmidt *et al.* 1996].

The possibility of there being a pattern language in the sense that Alexander used the term for software is a controversial one, and one that the OE&M group initially embraced with some scepticism. The basis of Alexander's proposed partnership between the inhabitants of buildings and the architect-

builder was that the real 'experts' of living buildings were the people who lived in them and worked in them. The pattern language was designed to generate 'living' architectures through complex emergent behaviour. The difficulty is in identifying the analogue to buildings architecture in software development. Nearly all human beings interact with buildings, whether living in them in or working in them, most of the waking minutes of every day of their lives. Software developers simply do not interact with software in the same way, although Gabriel speaks of the 'habitability' of source code [Gabriel 1996, p. 11].

Coplien suggests that the analogue is to be found, rather, in organization and that this is true not just for software developers, but for all professions [Coplien 1997, p. 243]. What was plain from our very first extended experience of the use of patterns for migrating legacy systems was that the patterns we were finding and using were highly interconnected. The use of one pattern seemed to set the context for the use of another in many instances. In other words our catalogue of patterns was exhibiting characteristics one might expect of an Alexandrine pattern language. It was at that point (the end of the second project) that the OE&M group coined the acronym ADAPTOR (Architecture Driven and Patterns-based Techniques for Object Re-engineering) for what was designated a candidate, embryonic pattern language. ADAPTOR is described as embryonic because:

- it is still evolving.
- it is not yet comprehensive in its coverage of the issues of legacy migrations.
- despite the successful use of its patterns to date, its generative character is not yet proven.

In its history to date there is more than a suggestion that software architecture may yet turn out to be the true analogue of the architecture of buildings in Alexander's language. The OE&M group is actively researching the theoretical aspects of patterns as possible explicators of the non-discursive aspects of construction, as discussed above. If true, this suggestion is not a complete refutation either of Gabriel's view or of Coplien's, but perhaps goes

some way to reconciling them. Software architecture, though not reducible to source code is clearly strongly related to it, while Conway's Law already establishes the parallelism between software architecture and the organization of software developers.

THE FOUR COMPONENTS OF A PATTERN LANGUAGE DESCRIPTION

The patterns in ADAPTOR contain four components (see Figure 1):

1. the abstract pattern itself (the problem-solution-context triple);
2. the pattern template to which its description conforms;
3. the pattern writers' workshops and other peer reviews through which it evolves; and
4. the pattern map (see Figure 2 for a high-level map of the ADAPTOR language), which describes its relationship to other patterns in the language.

Most descriptions of patterns in the public domain include only the first two components, but all four are necessary to the successful discovery, refinement, and use of software patterns.

The pattern template is important. It presents a standard structure in which the pattern documentation should be written, and in which the user of a pattern will expect to discover its contents. No one single standard pattern template exists. The Alexandrine [Alexander *et al.* 1977], Coplien [Coplien & Schmidt 1995] and Gang-of-Four [Gamma 1995] templates are the ones most well known in the public domain but, as mentioned above, templates can be company-specific or even project-specific. For this reason, and because ADAPTOR reuses patterns and even pattern languages that originated elsewhere, ADAPTOR patterns have more than one form. In the public domain, they use a form based on Coplien's [Coplien 1997]. This form is used in the Appendix to this paper, but in the catalogues of the companies in which they originated they have a form based on an in-house template. In addition, they are collected and indexed as 'thumbnails'. This form, too, is shown in the Appendix.

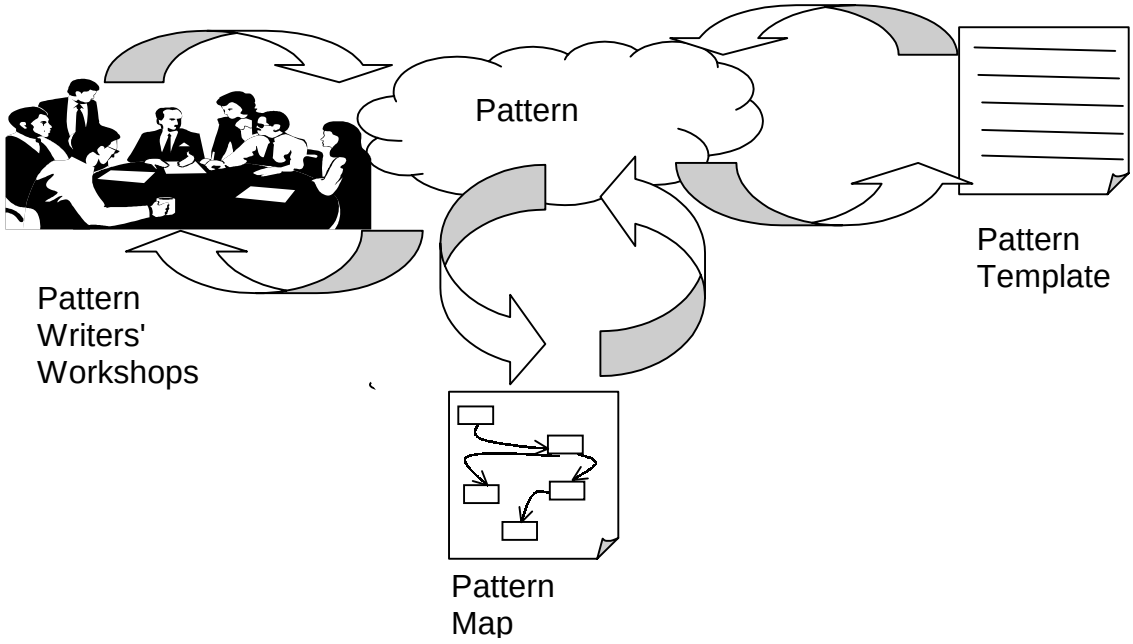

Figure 1 The Four Components of a Pattern Language Description

Patterns, when first drafted, enter the ADAPTOR language as candidate patterns. It is necessary for them to go through a formal review and redrafting experience that is common throughout the patterns movement: the pattern writers' workshop. The sole purpose of the pattern writers' workshop is to improve the pattern as a piece of literature. Its workings are fully described in Rising [1997]. A pattern writers' workshop is not a validation panel, which may take place separately. The potentially lengthy social process of peer reviews and validation which a pattern goes through before the 'candidate' prefix is dropped is an essential requirement for pooling the experiences of many developers, so as to pitch the pattern at the right level of abstraction.

AN OVERVIEW OF THE ADAPTOR PATTERN LANGUAGE

The ADAPTOR patterns can be categorized in a number of ways. They include newly discovered patterns abstracted from the four telecommunications projects, together with some seen in evidence elsewhere. They also include patterns already in the public domain, some in design patterns catalogues [e.g.,

Gamma 1995, Buschmann 1996] and others in process pattern languages that originated elsewhere [e.g. Coplien 1997]. The existing collection of patterns garnered from these sources already provides a high-level view of an overall process for legacy system migration. In particular the building of the analysis model is fairly well covered, together with some hooks into high-level and detailed design both of the software and the organization that supports it.

Figure 2 presents a high-level view of some of the existing patterns and the links between them. The map is a two-dimensional grid. The x-axis divides the patterns into technical patterns, process patterns and organizational patterns; the y-axis into concept model, high-level software architecture, and detailed design. The patterns with bold-face names and boxes are newly discovered patterns, the others already exist in the public domain in other sources. The arrows proceed from a pattern that sets the context for the use of the one it points to.

The map, though it shows only a small number of the ADAPTOR patterns, demonstrates their interconnectivity, and even at this stage of the research, a limited generativity. *Buffer the System with Scenarios* maps out alternate future contexts for the system's use. This, together with *System Composite*, sets the background for use case modelling to define the interfaces of the system components, which are themselves possibly reflections of the *Shamrock* pattern (described below). These interfaces can be represented by the *Façade* pattern, and/or a specialization of it, *Semantic Wrapper*. Both patterns mandate the modelling of semantically rich business components*. Conway's Law* now comes into operation, with each *Façade* being assigned to a team leader following the *Code Ownership* pattern.

By now the software architecture (the chosen separation of concerns) has helped deliver the outline, the 'scaffolding' if you will, both of the software system and of the team that is supposed to maintain it. The use of each pattern sets the context for the next one, without unnecessarily constraining the way in which it will be utilized. The next steps largely concern the detail of the implementation abstractions, which will, of course, include legacy code. At the moment the language only deals with the interfaces of those abstractions (via *Semantic*

Wrapper). The next stage is to fill out the language with patterns that address these issues.

If, as new patterns are added, the language continues to express the same level of generativity that can be seen here, then it may become possible to speak realistically about a pattern language for object migration. Such a language seems sure to include all of the kinds of patterns shown in Figure 2, with strong links to other pattern languages, and perhaps a number of mini-pattern languages embedded within it.

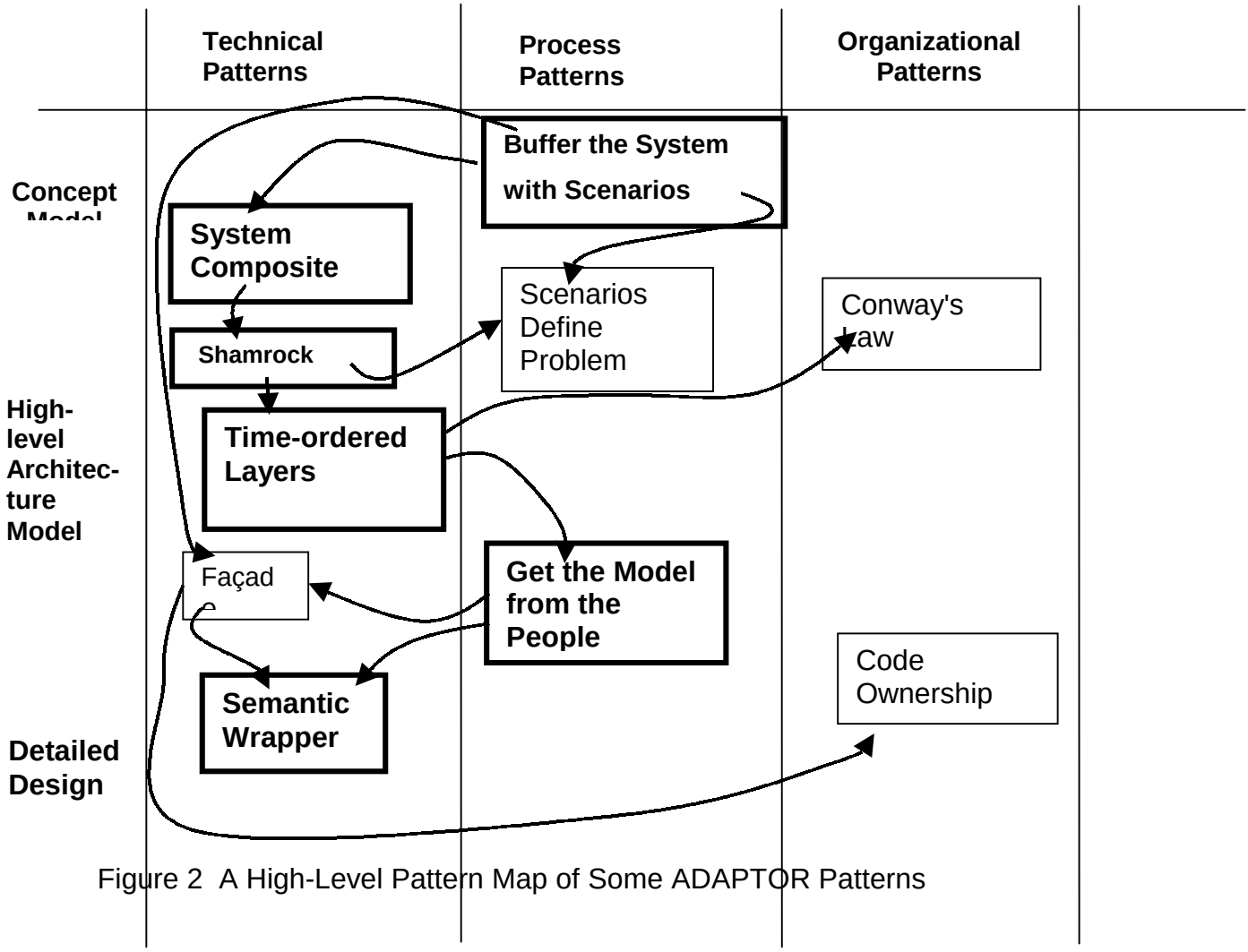

Figure 2 A High-Level Pattern Map of Some ADAPTOR Patterns

V. A CASE STUDY REVISITED

The ADAPTOR language is currently being evaluated in the CAD/CAM sector and also in the defence industry, as well as in a fifth project in the telecommunications sector. Commercial confidentiality agreements prevent disclosing detailed information about these projects, but one way of validating the patterns as they emerge is to run them against past projects, to see if evidence of their use can be found there. This is a form of pattern mining [DeLano 1997]. The same technique can be used to illustrate the utility of the ADAPTOR patterns by examining the one migration project listed in Table 2 that is already in the public domain. British Telecom's (BT) future-proof value added tax (VAT) processor project [Freestone 1996] used the architectural approach described above before it was cast into a patterned form.

THE BUSINESS NEED FOR THE FUTURE-PROOF VAT PROCESSOR

BT owns a customer service system which maintains information on each of its estimated 32 million customers (both private household and business) and the many products and services which it supplies. Its historical monopoly on telecommunications in the UK means that BT also happens to be the largest single collector of VAT on behalf of Customs and Excise. Telephone bills comprise two parts: an element for rental which is charged in advance typically on a quarterly basis, and an element for call charges which is made in arrears. While VAT was either zero-rated or charged at a single, constant rate (15% for a long period in the early 1990s) its collection through telephone billing did not constitute a major problem. But as recession loomed in the mid-1990s the Conservative government in the UK made two decisions that dramatically altered the situation. First, it changed the standard rate of VAT to 17.5% and then it introduced VAT on fuel at half that rate, raising a political furore about imposing a tax on an essential requirement for old age pensioners (OAPs).

Although BT was legally entitled to charge the new rate of VAT for both rental and call charges in its next quarterly bills, even if part of the period covered was when VAT was at a lower rate, its attempt to do so was a public relations

disaster at a time when the company was facing increased competition from cable companies and others. BT immediately committed itself to rebating its customers with the difference, and, indeed, fully implemented its promise. However, the administrative cost was huge, largely because both the procedures for calculating VAT and the data such calculations needed were scattered throughout the customer service system. This system was already 12 years old and had been subjected to major upgrades eight or nine times in each year of its life.

The government's decision raised the prospect of further, perhaps rather frequent VAT changes as a tool of monetary policy. Worse still, the political argument about VAT on fuel not only meant that different products might attract different rates of VAT, but also meant that different types of customer might be charged differently (e.g. exemption of OAPs). It was decided that a future-proof, 'one-stop shop' VAT processor was required, and that it should be crafted as a reusable software component in order to test the feasibility of migrating the customer service system incrementally to an object-based architecture.

Freestone and Wezeman [1996] of BT described this first, highly visible and very successful project as a three-phase process for object migration. A number of general guidelines for the migration of large-scale legacy systems were abstracted from this small subset of the customer service system. The following stages were involved:

- the creation of a Smalltalk prototype to gather requirements and model the functionality of the processor.
- the mapping of the abstractions modelled into MVS/COBOL (the current and target implementation technology of the customer service legacy system).
- the stubbing out of references to VAT in the legacy system, and the installation of the processor as a component within it.

PATTERNS USED IN THE VAT FUTURE-PROOF PROCESSOR

In retrospect, a number of the ADAPTOR patterns can readily be discerned in the migration. The ADAPTOR pattern *System Composite,* for example, treats any software system as a recursive aggregate of arbitrarily sized components. It

does not mandate any particular physical or logical characteristics of a component, other than it exists in a composition. This approach gives maximum freedom to the developer to impose an appropriate separation of concerns on the system. It also implies that a large-scale system can be treated as if it were a primitive component and vice versa. This understanding frees the developer to utilize the same requirements gathering and modelling techniques to describe any part of a system that could be used to describe the system as a whole.

In the case of the future-proof VAT processor, use case modelling techniques were used to capture an understanding both of the way VAT was currently being collected, and to explore the ways it might change. These requirements gathering activities are examples of two process patterns: one already in the public domain, the other only recently added to the ADAPTOR language. The utilization of use cases to capture the 'as is' requirements reflects pattern 22, *Scenarios Define Problem* in Coplien's organization and process pattern language [Coplien 1997], but the engagement of BT's internal VAT experts in modelling potential future scenarios reflects the *Buffer the System with Scenarios* pattern. This pattern is included in full in the Appendix to this paper.

The initial modelling of the future-proof VAT processor was done without reference to the existing representation of relevant data in the customer service system, or indeed to any part of the legacy system other than to note that it held information about customers and products. This scoping of the analysis model so that it captured the key abstractions of the problem space and modelled them separately and independently of any implementation concerns reflects the *Shamrock* pattern of the ADAPTOR language. A thumbnail of this pattern can be found in the Appendix .The stylized 'three-leaf shamrock', which gives the pattern its name, is shown in Figure 3.

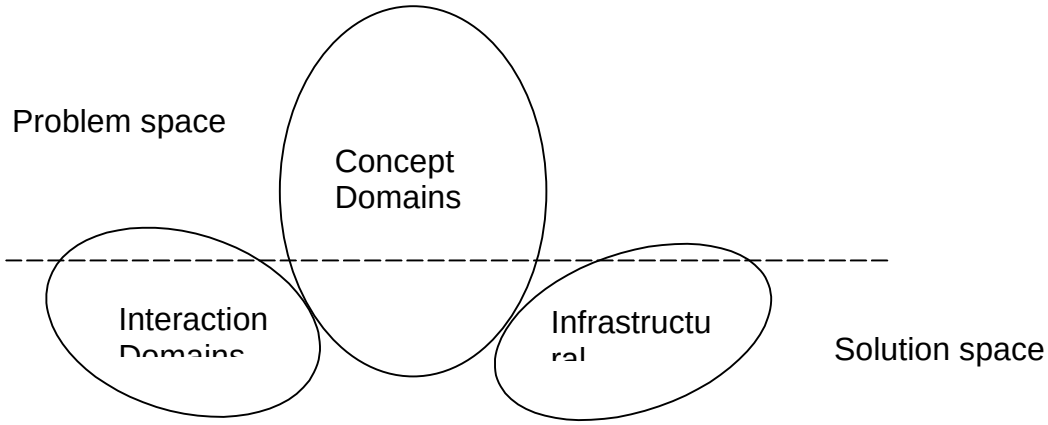

Figure 3 The Shamrock Pattern Illustrated

Each leaf reflects a common type of architectural domain that can be found in object-based or component-based systems, particularly information systems. In other words, these are the kinds of concerns that are most typically separated at the top level. The central leaf represents the problem space domain or domains (depending on whether the problem space itself needs to be partitioned logically). Its position in the illustration indicates that these domains form a conceptual model which captures the key abstractions in the problem space. They map to objects or components that directly represent them in the solution space (so-called 'business objects' or 'business components'). A business user would normally be able to recognize the abstractions contained in these domains, and the relationships between them. The other two leaves contain domains that are relevant to the software solution primarily, rather than the problem space. They are the infrastructural domains (e.g. for concurrency, persistence, distribution, etc.) and the interaction domains (e.g. GUIs or machine-to-machine interfaces). Each one of these types can include one to many domains itself, as needed, and is characteristically the concern of the designers of the software.

The point of the pattern is to underline the need to separate these concerns as cleanly as possible, by dealing with the issues of on-screen presentations separately from the key concepts, for example. This separation is exactly what was done in the future-proof VAT processor case study. By postponing

consideration of the *representation* of the key VAT abstractions in software (to phase 2), and of their implementation and interfacing to other components in the customer service system (phase 3), the development team was able to capture the essence of the business problem by working with the users. The software development was driven by the business instead of the other way round.

A number of other patterns are reflected in the case study, but here we will mention just two more. The prototype used to capture the business problem and validate the proposed solution was a Smalltalk prototype that made full of use of its encapsulation, inheritance, polymorphism, and other features. Most of these language features do not exist in MVS/ COBOL, the target implementation technology. A major design decision was to create Policy objects to abstract away the complexity of the different permutations for taxing customers and products, and even for rounding figures after calculation. In retrospect it is clear that these objects played roles which are recognizable in the *Mediator* and *Strategy* patterns [Gamma 1995]. More generally, both the prototype and the live solution made use of a pattern called *Time-Ordered Layers* (see Appendix) even though they were necessarily different implementations. This pattern from the ADAPTOR language calls for components with similar change rates to be grouped together, with the slower changing 'layers' constraining the faster ones to limit the 'shearing effects' in a system of components being changed with different frequencies.

V. CONCLUSION

The experience of four successful migration projects in five years clearly demonstrated the importance of

- focusing on software architecture (the partitioning of a system according to a specific separation of concerns) and on
- achieving a strong correspondence between the key abstractions in the problem space and software components in the solution space.

Flexible architectures that can support reuse and at the same time be flexible to business requirements must necessarily be shaped by the vocabulary

of the problem domain. To do so requires the conscious and explicit use of object modelling methods.

Expertise in shifting legacy systems to new paradigms is buried in the folklore of software engineering. Patterns offer a way of capturing and communicating best practice in migration projects, just as much as they do for object-oriented design. A number of such patterns have been accepted and are now in use in the telecommunications sector. 'Migration patterns' seem to demonstrate a high connectedness and interdependence, including between design, process, and organizational patterns. By documenting such patterns, the host organization captures elements of the configurational aspects of design which otherwise go unreported. The documentation makes explicit knowledge that may be critical for future maintainers in understanding why one of a number of possible design solutions was chosen. It also shortens the learning curve of less experienced developers by providing exemplars of best practice. Finally, by utilizing such patterns as guidelines, the considerable risk involved in each subsequent migration of a legacy system is reduced substantially.

ADAPTOR is an evolving, candidate pattern language for the migration of legacy systems. It continues to grow and evolve, exhibiting some of the characteristics of an Alexandrine pattern language. As yet it has neither the coverage nor the generative power to be considered a full pattern language. Research into the theoretical aspects of patterns and pattern languages, in particular their relationship to theories of 'social knowledge' and non-discursivity in design, is still being carried out by the author, and by De Montfort University's OE&M group. In the meantime, further live industrial-strength case studies and projects will fully test out and either validate or invalidate the potential of ADAPTOR as a genuine pattern language. Its catalogue of existing patterns, considered as largely 'stand-alone' patterns, continues to be effective in its own right and is being applied to sectors other than the telecommunications industry in which it originated.

Editor's Note. Christopher Holland served as Editor for this article. It is part of the Focus Issue on Legacy Systems and Business Process Change The article was fully refereed. It was received on

February 25, 1999 and published on July 30, 1999. The manuscript was with the author for approximately 3 weeks for 2 revisions.

LIST OF ACRONYMS

ADAPTOR	Architecture-Driven and Patterns-based Techniques for Object Re-engineering
BPR	Business Process Re-engineering
BT	British Telecom
CAD/CAM	Computer Aided Design/Computer Aided Manufacturing
CBD	Component Based Development
OAPs	Old Age Pensioners
OE&M group	Object Engineering and Migration group
OT	Object Technology

REFERENCES

Alexander, C. (1964) *Notes Towards a Synthesis of Form*, New York: Oxford University Press.

Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel (1977) *A Pattern Language*, New York: Oxford University Press

Appleton, D. (1983) "Law of the Data Jungle", *Datamation*, October, 29(10): 225-30

Booch, G. (1991) *Object-oriented Design with Applications*, Redwood City, CA:Benjamin/Cummins

Brooks, F.P. Jr. (1986) "No Silver Bullet - Essence and Accident in Software Engineering" in H.-J. Kugler (ed.) *Information Processing '86*, Amsterdam: Elsevier Science (North Holland), pp. 1069-76.

Brooks, F.P. Jr. (1995) " 'No Silver Bullet' Refired", *The Mythical Man Month*, 2nd ed. Reading, MA: Addison Wesley

Bucken, M. (1992) "Travellers Preserving Programming Pillars; Insurance Giant Pushes Reengineering Pilot Successes to IS Skeptics", *Software Magazine*, October, 12 (14): 48

Buschmann, F. R. Meunier, H. Rohnert, P. Sommerlad and M. Stal (1996) *Pattern-Oriented Software Architecture: A System of Patterns*, Chichester, England: John Wiley and Sons

Chen, P. (1989) "Entity Relationship Model: Towards a Unified View of Data", *ACM Transactions on Database Systems*, 1 (1)

Chikofsky, E. and J.H. Cross II (1990) "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, January, 7 (1): 13-17

Chomsky, N. (1957) *Syntactic Structures,* The Hague: Mouton

Connall, D. and D. Burns (1993) "Reverse Engineering: Getting a Grip on Legacy Systems", *Data Management Review,* October: 24-7

Constantine, L. and E. Yourdon (1976) *Structured Design*, Englewood Cliffs, NJ: Yourdon Press, Prentice Hall

Cook, S. (1994) "Analysis, Design, Programming: What's the Difference?" in A.J. O'Callaghan and M. Leigh (eds) *Object Technology Transfer*, Henley-on-Thames, England: Alfred Waller, pp. 55-64

Cook, S. and J. Daniels (1994) *Designing Object Systems*, Englewood Cliffs, NJ: Prentice Hall

Coplien, J.O. (1996) *Software Patterns*, New York: SIGS Books

Coplien, J.O. (1997) "A Generative-Development Process Pattern Language" in L. Rising (ed.) *The Patterns Handbook*, New York: SIGS Books, pp. 243-300

Coplien, J.O. and Schmidt D. (1995*) Pattern Languages of Program Design*, Reading, MA: Addison Wesley

DeLano, D.E. (1997) "Patterns Mining" in L. Rising (ed.) *The Patterns Handbook*, New York: SIGS Books, pp. 87-96

Farmer, R.W., A.J. O'Callaghan, L.T. Harries and N.K. McBride (1996) "Dealing with Legacy Systems and Legacy Culture", Workshop, *Object Technology '96* Oxford, England

Freestone, D. and C. Wezeman (1996) "Object Lesson: Changing Legacy Systems" in A.J. O'Callaghan (ed.) *Practical Experiences in Object Technology*, Cheltenham, England: Stanley Thornes

Gabriel, R. P. (1996) *Patterns of Software - Tales from the Software Community*, New York: Oxford University Press

Gamma E., R. Helm, R. Johnson and J. Vlissides (1995) *Design Patterns: Elements of Reusable Object-Oriented Software* Reading, Mass: Addison-Wesley

Graham, I. (1995) *Migrating to Object Technology*, Wokingham, England: Addison Wesley

Graham, I. (1998) *Requirements Engineering and Rapid Development*, Wokingham, England: Addison Wesley

Hammer, M. and J. Champy (1993*) Reenginering the Corporation: A Manifesto for the Business Revolution*, New York: Harper Collins

Harries, L.T. (1996) "Towards a Catalogue for Patterns", Confidential Technical Report, Object Engineering & Migration Group, School of Computing Sciences, De Montfort University, Leicester, UK

Hillier, B. (1996) *Space is the Machine*, Cambridge: Cambridge University Press

Jacobson, I. and F. Lindstrom (1991) "Re-engineering of Old Systems to an Object-Oriented Architecture", *Proceedings of OOPSLA '91* New York: ACM Press

Jones, T. (1994) *Assessment and Control of Software Risks*, Englewood Cliffs, NJ: Yourdon Press, Prentice Hall

Kuhn, T.S. (1970) *The Structure of Scientific Revolutions*, second edition, Chicago, IL: University of Chicago Press

Lano, K. and H. Houghton (1993) *Object-oriented Specification Case Studies*, Hemel Hempstead, England: Prentice Hall

Lerner, M. (1994) "Software Maintenance Crisis Resolution: The New IEEE Standard", *Software Development*, August, 2(8): 65-72

Liao, S.Y., P. Shao and W.H. Tsang (1998) "Experience Report: SSADM-Designed System to Object-Oriented System*", Journal of Object-Oriented Programming*, February: 38-48

Martin, J. and J. Odell (1998) *Object-Oriented Methods: A Foundation (UML Edition)*, Englewood Cliffs, NJ: Prentice Hall

Meyer, B. (1988) *Object-oriented Construction*, first edition, Englewood Cliffs, NJ: Prentice Hall

Meyer, B. (1996) *Object-oriented Construction*, second edition, Upper Saddle River, NJ: Prentice Hall

O'Callaghan, A.J. (1994) "Object Skills: Why, What and How" in A.J. O'Callaghan and M. Leigh (eds) *Object Technology Transfer*, Henley-on-Thames, England: Alfred Waller, pp. 3-30

O'Callaghan, A.J. (1996) "Legacy Systems, Legacy Culture: Crossing the Great Divide", *Proceedings of ObjectExpo Europe*, September

O'Callaghan, A.J. (1997) "Object-oriented Reverse Engineering", *Application Development Advisor*, 1 (1): 35-9

O'Callaghan, A.J. (1998a) "Perverse Engineering", *Application Development* Advisor, 1 (5): 56-60

O'Callaghan, A.J. (1998b) "ADAPTOR: A Pattern Language for the Reengineering of Systems to Object Technology", *IEE Informatics Division Colloquium on Understanding Patterns and their Application to Systems Engineering*, Digest No. 98/308, London, 28 April

Rising, L. (ed.) (1997) *The Patterns Handbook*, New York: SIGS Books

Schmidt, D., M. Fayad and R. Johnson (1996) Guest editorial *Communications of the ACM Special Issue on Software Patterns* 39 (10)

Selic, B., G. Gullekson and P.T. Ward (1995) *Real-time Object-oriented Modeling*, New York: Wiley

Sprott, D. and L. Wilkes (1998) *Component-Based Development: Application Delivery and Integration Using Componentised Software,* Hull, England: Butler Group

Ulrich, W. (1994) "From Legacy Systems to Strategic Architectures", *Software Engineering Strategies*, March/April, 2(1):18-30

Wirth, N. (1971) "Program Development by Stepwise Refinement", *Communications of the ACM*, 14(4): 221-7

APPENDIX
Selected Excerpts from the ADAPTOR Patterns Catalogue

The patterns below are reprinted in the form in which they appear in the ADAPTOR pattern catalogue.

BUFFER THE SYSTEM WITH SCENARIOS

Problem: If business requirements shape software architecture, but the business context is volatile, how do you start constructing a software architecture?

Context: Main functional requirements have been specified, possibly in an implied rather than explicit, overall business context stretching ahead over time.

Forces
All significant software systems are predictions: all predictions are wrong.
- Prioritization of requirements is typically dictated, *in the final analysis*, by the business context the software system serves, but in the first analysis there are usually a host of hidden assumptions underlying the business perspective itself.
- A software system built to support only one perspective for the business will almost certainly prove to be brittle to change.
- A software system that tries to meet the requirements of all possible scenarios will almost certainly suffer 'analysis paralysis' and will be too complicated and/or inefficient to deliver and use.

Solution
Draw on the expertise of business planners, marketeers and domain experts to extract 3 -7 'alternative' business perspectives to the 'main' one around which the company's operations are being planned. Name the scenarios, write them down in concise and precise terms and extract the key impacts that the eventuality of each scenario would have for the system. Develop an optimal architecture with sufficient flexibility to allow it cope with any of these scenarios should they develop for real.

Resulting Context
A minimum gain is that the environment which sets the context for the use of the system is better understood by the developers including, crucially, an understanding of the key factors impacting upon its business scope. This lays the basis for the development of a more adaptable software architecture, flexible to major business changes. The use of this pattern sets the context for the use of the *Time-Ordered Layers* pattern.

Rationale
Scenario buffering is an established technique in the architecture of the built environment for scoping new office buildings and major refurbishment. Enabling techniques such as use cases, role-modelling (e.g. OORAM) and task scripts (e.g. in SOMA) are already wide spread in object-oriented software development, and there is already existing usage of adapted versions of these techniques (e.g. 'Change Cases') to explore future scenarios. *Buffer the System with Scenarios* is related to *Scenarios Define Problem* in Jim Coplien's *Generative Development-Process Pattern Language.*

(Note: the pattern template used here follows the form of Coplien's *Generative Development-Process Pattern Language* referred to above.)

TIME-ORDERED LAYERS

Problem: How is the high-level structure of a system best organized for adaption in the long term?

Context: The business and/or technical environment into which the system is to be deployed is understood to be volatile over time, and will require as yet unspecified changes to be made to the system.

Forces
- Change is the only constant in a long-lived system.
- Structural stability is a requirement of long life.
- Different aspects of the system are impacted by different kinds of change requirements (e.g.
business, technical, environmental, legal, etc.).
- Different elements of a system change at different rates.
- Change effects need to be localized to minimize cost and effort "ripple effects".

Solution
Organise the system into layers such that the components of each layer have similar lifespans and/or change rates. Each layer should be distinguishable from the others on the basis of the expected change rates of its components. Design the 'permanent' and slowest-changing layers first and proceed in a time-ordered manner, moving to the next slowest layer and so on.

Resulting Context
A system layered according to the different change rates of its components localizes the effects of change. "Slower-moving" layers constrain the design of those with faster change rates, resulting in a system which is stable but flexible both to different kinds of change, and to changes with differing frequencies.

Rationale

Time-ordered layering is a well-established technique in office building design enabling the flexible utilization of buildings with say, sixty-year life spans, so that services (electrical wiring, plumbing) can be overhauled every seven years on average, but office space altered as frequently as daily if required (Duffy 1990). It is based on the observation, also seen in biology (O'Neill 1986) that elements with similar life-spans often form cohesive systems and subsystems which respond to similar kinds of change forces. In the development of commercial information systems a well-known precept is to design the data model first, on the basis that it is less volatile than process (e.g. Howe 1983). Similarly, the well-known three and n-tiered client/server architectures can be seen to observe principles of time-ordering.

References

Duffy, F. (1990) "Measuring Buildings Performance", *Facilities*, May

Howe, D. R. (1983) .*Data Analysis for Database Design*, Edward Arnold, London, England

O'Neill, R.V. *et al*. (1986) *A Hierarchical Concept of Ecosystems,* Prentice Hall, Englewood Cliffs, NJ

Thumbnails of Some Key ADAPTOR Patterns

The patterns below are presented in the form in which they appear in the ADAPTOR pattern catalogue.

System Composite. This pattern is an analogue of the *Composite* design pattern in the Gamma catalogue. It is the most fundamental of the patterns – one which creates the context for the other patterns. System Composite views all systems as recursive aggregates being made up of subsystems of components and connectors. These subsystems can themselves be treated as systems in their own right. As with the Gamma *Composite* the power of the pattern is that these aggregates can be treated in the same way as system primitives. This permits system-level modelling techniques to be used at arbitrarily recursive depths in any large-scale system. *System Composite* is a pattern discovered by the Object Engineering and Migration group at De Montfort University.

Scenarios Define the Problem. This pattern exists in Jim Coplien's generative software development pattern language. The pattern describes the utilization of Use Cases or Task Scripts to capture interractions between external "actors" and the system to both capture functional requirements and drive the extraction of candidate classes etc. Actors can be humans, external systems or, applied recursively, other subsystems. The significance of using this pattern in the context set by *System Composite* is that it opens the way for Object-Oriented Analysis modelling of the problem space, including that part of it occupied by the legacy system. The use of similar techniques to forward engineering of object systems is fundamental to the approach described by ADAPTOR.

Get the Model from the People. This is a process pattern discovered in migration work done at BT by the Object Engineering and Migration group at De Montfort University. It was published at the first TelePlop (Telecommunication Pattern Languages of Programs) workshop at OOPSLA '96. It focuses on the notion that system maintainers hold in their heads and in their work culture valuable knowledge about the legacy system which is not held documented elsewhere. *Get the Model from the People* is actually itself the entry point into a small, self-contained pattern language.

Shamrock. This pattern is a recent addition to ADAPTOR. It is based on the observation that most object-oriented information systems rest on a domain structure in which the domains can be classified into three kinds: concept domains, interaction domains and infrastructure domains. In conjunction with *Scenarios Define the Problem* and *Get the Model from the People,* this pattern can be used in early analysis to shape the high-level topology of the migrated system.

Façade. Façade is a Gamma pattern which describes an object which sits on the logical boundary between two subsystems. It presents a single interface of a

subsystem to its clients, delegating requests for services of the subsystem to the actual objects (or other software entities in the case of a legacy system) which implement the requested behaviours. *Façade* is fundamental to the migration of legacy systems, allowing as it does software on either side of the façade to evolve independently.

Semantic Wrapper. This pattern can be considered to be one that implements *Façade* in a legacy system context. The basic notion is that classes which exist to access legacy code should differ from other objects only in their implementation details. The interface they present to the rest of the system should capture and present abstract behaviours which have semantic content.

ABOUT THE AUTHOR

Alan O'Callaghan is Senior Lecturer at De Montfort University, Leicester in the United Kingdom. He has been a national committee member of the British Computer Society's Object Oriented Programming and Systems (BCS OOPS) specialist group since 1993, and is also on the Pattern Languages of the United Kingdom (PLUNK) co-ordinating committee. He has edited two books on object technology, and is a regular columnist for *Application Development Advisor* on the migration of legacy systems. His research interests include object technology, software architecture, software patterns and the migration of legacy systems to componentized software.

LETTERS TO THE EDITOR

From: Ashley D. Lloyd
Management School
University of Edinburgh

To the Editor of CAIS:

I believe there is value in reading our paper *BUSINESS PROCESS AND LEGACY SYSTEM REENGINEERING: A Patterns Perspective* (by Lloyd, Dewar, and Pooley, Communications of AIS Vol. 2, No. 24) together with O'Callaghan's paper *Migrating Large-Scale Legacy Systems to Component-based and Object Technology: The Evolution of a Pattern Language.* Both papers promote Patterns as a means of capturing knowledge about the reengineering of legacy computer systems, both view the legacy issue as primarily a business problem, and both recognise that different information is required by different participants in the reengineering process.

The papers however differ in a number of important respects. O'Callaghan's definition of a legacy system is more restrictive and focusses on large-scale systems. Our definition could apply to a system of any size, age, or value, though it does recognise that reengineering is not the only solution to a legacy problem. Our paper also recognises that a 'large-scale' problem can emerge through the 'intra-structure' coupling of smaller legacy systems – a coupling which for many smaller systems is the business process itself.

O'Callaghan recognises the need to apply patterns to non-technical areas, but does not explicitly support communication of concepts or priorities between the 'business' and 'technical' domains, or between different hierarchical levels in an organisation. We feel that such communications is critical in assisting balanced decision making and address this problem through embedding generic concepts of competitive advantage within the patterns themselves - an area that is discussed in detail within our paper.

It should be remembered that both papers recognise the emergent nature of the patterns field, and that part of the validation process for patterns and the methods by which they are recorded and used is publication. Neither paper claims to provide answers to all the questions they raise, and in this light, we welcome comments on either paper.

Response from:
Alan O'Callaghan
DeMontfort University

To the Editor of CAIS:

CAIS is to be congratulated on recent publication of two papers focusing on the use of patterns in legacy systems re-engineering. I welcome in particular *Business Process And Legacy System Reengineering: A Patterns Perspective* (by A.Lloyd, R. Dewar and R. Pooley, CAIS Vol 2. no. 23) as a complement to my paper *Migrating Large-Scale Legacy Systems To Components And Object Technology: The Evolution of a Pattern Language* (Vol.2 no. 3). Together they expose an exciting new area of research for Computing Scientists, Software Engineers, Information Systems Engineers, and Business Analysts alike. The papers hold in common, I believe, a concern for developing communication between the different stakeholders involved in a legacy re-engineering project; in advocating the primacy of assessing business need both in assessment of the problem and the provision of the solution; and in utilising a specifically "Alexanderian" approach to the identification of potential patterns.

In the comments on my own paper that Ashley Lloyd presented in his letter to the editor the following differences are highlighted:
- my own definition is of a legacy system is more restrictive and focuses on large-scale systems
- the Lloyd, Dewar and Pooley paper recognises that, presumably at enterprise-level, large-scale legacy problems can emerge as a result of coupling between smaller legacy systems which make up a business process
- the O'Callaghan paper does "not explicitly support the communication of concepts or priorities between the business and technical domains"

Based on a reading of the paper all three points are valid, though I would add here the qualification that the ADAPTOR pattern language as a whole does attempt to deal with issues in the last of them, albeit in a different way from the Lloyd et al. paper. A candidate pattern 'Archetype' which is not reflected in my paper, but is an ADAPTOR pattern, stresses for example the need for the core building blocks of a software architecture to reflect business abstractions. The first two points Lloyd makes, are I think, more important because they reveal the key differences in the contributions being made by the two research projects. There is an emphasis on managerial issues in the Lloyd et al patterns which can be seen in the'Middleware', ' War Room' and 'Work Shop' patterns in particular.

These are illustrated by issues that can emerge when dealing with vendors of software products. In other words the scope of their patterns includes the tactical problems that users have to deal with in dealing with a system, or a number of systems that make up a legacy. Such patterns, whose validity I fully accept, are beyond the existing scope of my own work. ADAPTOR has sought to concern itself with the strategic issue of how a software architecture can be developed, in the context of existing IT investment such that it can be more generally be made flexible to business change. The ADAPTOR patterns are therefore certainly more restrictive in scope in the sense of this focus on the software architecture, but are simultaneously more broad in the sense that they deal with this issue of architecture at a strategic level. The difference in the working definitions are largely explained by this difference in focus. It seems likely that any mature pattern language worthy of the name will need to include elements from both these axes of research if it is to be sufficiently comprehensive to be truly generative.

Lloyd is quite right to remind your readers of the emergent nature of the patterns field. Within the 'patterns community' some important debates are beginning to take place. One of these has to do with the significance of patterns as "stand alones" versus pattern languages. The September/October issue of IEEE Software, for example, focuses on Software Architecture and includes a Guest Editorial by Jim Coplien. He suggests that the advocates of pattern languages implicitly embrace a notion of 'architecture' which evolves through piece-meal growth and are therefore hostile to the received wisdom in IT of architectures defined as master plans or detailed blueprints contained in (often voluminous) design documentation. Alexander is not only quoted, but his speech to the ACM OOPSLA '96 conference in San Jose California is reprinted in full. From this perspective the 'Design Patterns' contained in the famous Erich Gamma et al. book might be considered "degenerate" because, valuable though they are, they are mainly pieces of reusable design structure. Because they are not elements of a language they are not, and cannot be, generative. There is no sense of architectural vision that unites them.

ADAPTOR explicitly embraces the idea that patterns need to be seen as part of a pattern language if they are to deliver their full potential. I personally accept the broad proposition that Coplien has put forward about the relationship between pattern languages and architecture. Pattern languages for software development will have to go beyond the scope of Gamma et al. design patterns (a point which is strongly made in both papers, incidentally), and seek a level of comprehensiveness (in scope) and interlinking which has not yet been achieved anywhere in our related disciplines. I hope CAIS readers will recognise that the two patterns papers are, taken together, ground-breaking contributions to that development. January 9, 2000